PRÁCTICA 2: ETIQUETADO

INTELIGENCIA ARTIFICIAL 2019 - 2020

AUTORES:

Dídac Piferrer Iglesias, 1497551
Marc Núñez Fuertes, 1492383
Allen Alarcon Jimenez, 1491223

GRUPO:

DL.15-DJ.17

Índice

Introducción de la práctica

La práctica consiste en la resolución de un problema de etiquetado de imágenes, se programarán una serie de algoritmos con los cuales podremos etiquetar un conjunto de imágenes por su forma y color.

Se hará uso del "Fashion Product Images Dataset" de Kaggle que es un repositorio de una base de datos donde habrá imágenes de baja resolución (60x80), de esta forma el tiempo de ejecución de los algoritmos será reducido.

Para hacer está práctica hemos utilizado IDEs como PyCharm y Anaconda para programar el código y ejecutarlo. Además, hemos hecho uso de guías de usuario tanto de Python como de las librerías utilizadas como numpy y también el conocimiento adquirido a lo largo de la carrera.

El objetivo final de esta práctica reside en programar todos los algoritmos para crear un programa en el que un usuario pueda hacer una pregunta de alguna pieza de ropa o color y que el etiquetador le conteste un conjunto de imágenes a partir de esta.

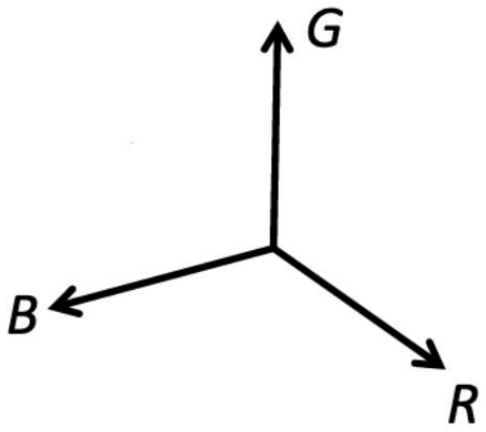
Para resolver el problema se han dividido los etiquetados en:

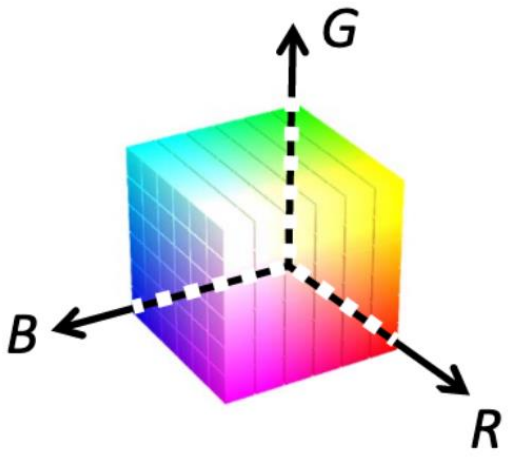
- Etiquetado automático de color (no supervisado)
- Etiquetado automático de forma (supervisado)

Etiquetado automático de color (no supervisado)

El objetivo de este etiquetador es que a partir de una imagen obtenga los colores de esta. Para implementar esto, debemos saber cómo se representa los colores a partir de una imagen, donde se trata de una matriz de NxMxC (fila x columna x canal).

La razón de que haya 3 canales son por los colores RGB (Red-Green-Blue) y de esta manera representamos un espacio de características.





A la hora de agrupar los puntos de colores, se hará uso del algoritmo de Kmeans, donde permite encontrar agrupamientos de puntos en una muestra dada. Los diferentes parámetros del algoritmo son:

- X: Conjunto de puntos que forma el conjunto de aprendizaje.
- K: Nombre de clases donde quiere dividir el espacio.
- Ci: Conjunto de puntos de la clase i
- CI_i^t: Centro de inercia de la clase al instante t.
- $d(\vec{x}, \vec{y})$: distancia entre dos puntos.

El algoritmo de Kmeans tiene 2 parámetros de entrada X y K, y está formado por un inicializador y un bucle.

El inicializador lo que hace es escoger puntos para declararlos los centros de inercia iniciales, el criterio de selección depende de las opciones con las que se inicialice el algoritmo. En el bucle, asociamos cada punto en una clase, en función de sus distancias que después se hará la media para saber el centro y se agruparan. El bucle termina cuando los centros de inercia sean iguales y que no cambien.

Funció k-means (X, k)

1. Escollir aleatòriament k punts de X per inicialitzar $CI_i^0, \forall i : 1 \dots k$
2. **Repetir**
   1. **Per** a (cada classe $C_i, \forall i : 1 \dots k$) fer
      1. $C_i = \{x \in X : \forall j \neq i, d(x, CI_i^t) \leq d(x, CI_j^t)\}$
      2. Calcular el nou centre: $CI_i^{t+1} = \left(\sum_{j=1}^{\#C_i} x_1^j / \#C_i \quad \cdots \quad \sum_{j=1}^{\#C_i} x_d^j / \#C_i \right)$
   2. **FPer**
   3. t++;
3. **Fins que** ($CI_i^t = CI_i^{t+1}; \forall i : 1..k$)
4. **Retornar**($\{CI_i^t\}_{i\,:\,1..k}$)

FFunció

Para hacer una estimación del número de clases, haremos uso de la Distancia Intra-Class, que Suma para todas las clases de la media de las distancias entre todos los pares de puntos de una clase.

$$D(C) = \frac{2}{m(m-1)} \sum_{j=1} \sum_{i=j+1} d(\vec{x}^i, \vec{x}^j) : \vec{x}^i, \vec{x}^j \in C, i, \text{j}: 1 \dots m$$

A la hora de hacer el código lo escribiremos en el fichero de Kmeans.py, donde implementaremos la clase Kmeans y han de tener estas funciones:

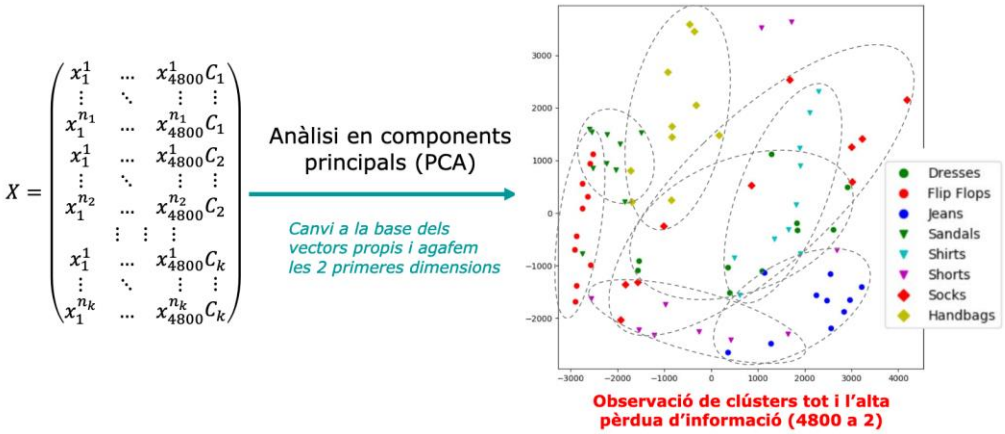
- **`_init_options:`** Indica los parámetros del funcionamiento del kmeans.
- **`_init_X:`** Inicializa los píxeles de la matriz X a partir los puntos RGB.
- **`_init_centroid:`** Inicializa los centroides y se implementan los centroides y los antiguos centroides.
- **`Distance:`** Calcula la distancia entre cada píxel y cada centroide
- **`Get_labels:`** Calcula el centroide más cercano de todos los puntos de X y asigna cada punto al centroide más cercano.
- **`Get_centroids:`** Calcula las coordenadas de centroides en base a las coordenadas de todos los puntos asignados al centroide.

- **`Coverges:`** Comprueba si hay una diferencia entre los centroides actuales y anteriores.
- **`fit:`** Ejecuta el algoritmo KMeans hasta que converge o hasta que el número de iteraciones es menor que el número máximo de iteraciones.
- **`withinClassDistance:`** Calcula la distancia Intra-Class.
- **`find_bestK:`** Indica la máxima k que se analizará.
- **`Get_color:`** Convierte los valores RGB en etiquetas de color.

Etiquetado automático de forma (supervisado)

En este caso lo que se busca es deducir la forma de la prenda a partir de la imagen.

En el momento de clasificar la ropa, pasaremos del espacio de 4800 dimensiones a 2 dimensiones mediante el análisis en componentes principales (PCA), en que más tarde los dividimos en 8 clases que son las piezas de ropa.



Un inconveniente de esto es que produce mucha pérdida de información que no hay suficiente dimensión para ver la separación de los clústeres.

Como los valores no presentan un modelo claro, implementaremos el clasificador del vecino-k más cercano (KNN), donde se basa en una función de decisión. Hay que tener en cuenta estas funciones.

- $d(\vec{x}, \vec{y})$: distancia entre los puntos x e y.
- Inserir(E,L): se añade un elemento E en una lista L.
- Ordenar_d(L): devuelve una lista ordenada según su campo.
- Primers_k(L,K): devuelve una lista L con los primeros k valores.
- Comptar(L,C): cuenta cuantos elementos de la lista L pertenece a la clase C.

El algoritmo se basa en calcular la distancia de un punto de que queremos clasificar a cada uno de los puntos de la muestra, de todas las distancias, lo ordenamos y nos quedamos con los k vecinos más cercanos. A partir de estos vecinos asignaremos la clase del punto en cuestión.

$Per\ (\vec{x}^j \in \boldsymbol{X})\ fer$
$\quad Llista = inserir([d(\vec{y}, \vec{x}^j), C_j], Llista)$
$fPer$
$Veins = Primers\_k(ordenar\_d(Llista))$
$Si\ (comptar(Veins, C_1) > comptar(Veins, C_2))$
$\quad \vec{y} \in C_1$
$Sinó$
$\quad \vec{y} \in C_2$
fSi

A la hora de hacer el código lo escribiremos en el fichero de KNN.py, donde implementaremos la clase KNN y han de tener estas funciones:

- **`_init_train:`** Función que se asegura que la variable train_data de la clase KNN, la cual contiene nuestro conjunto de entrenamiento.
- **`get_k_neighbours:`** Calcula los k vecinos más cercanos en cada punto.
- **`get_class:`** Comprueba cual es el etiquetado que ha aparecido con más frecuencia.
- **`predict:`** predice la clase a la que pertenece cada elemento en test_data.

Métodos de análisis implementados

Análisis cualitativo

Para realizar el análisis cualitativo hemos creado diferentes test que nos permitirán evaluar el comportamiento de los algoritmos desde diferentes puntos de vista.

Los test que hemos realizado son:

- Test_Retrieval_by_color
- Test_Retrieval_by_shape
- Test_Visualize_kmeans

Con tal de hacer unos experimentos más reales que nos lleven a un mejor análisis, hemos optado por analizar siempre todas las imágenes en cada test, excepto en el caso de Test_Visualize_kmeans. Cuando obtengamos las imágenes que, según nuestros algoritmos, cumplen con los requisitos devolveremos una cantidad aleatoria de estas para verificar si nuestro algoritmo ha acertado o no, y en el caso negativo, analizar por qué. Para poder implementar esa aleatoriedad hemos hecho uso de la librería random.

Todo y que los test conlleven más tiempo, hemos optado por esta decisión ya que para sacar con firmeza unos resultados a partir de tantas imágenes creemos conveniente optar por la aleatoriedad. Después de un seguido de test, podremos sacar más o menos unas conclusiones mediamente coherentes, sin embargo, solo seleccionando X imágenes, nos dejamos ciertos factores que consideramos que pueden entorpecer el análisis.

Test_Retrieval_by_color

Este test nos permite saber si nuestro algoritmo Kmeans acierta con las etiquetas de colores que nos devuelve, comparándolas con el GroundTruth que se nos proporciona.

La función recibe 4 parámetros:

- Una lista con todas las imágenes a evaluar.
- La "k" máxima con la que queremos que se ejecute nuestro algoritmo Kmeans.
- La petición realizada, en este caso, un color (string), o una lista de colores (list).
- El número de imágenes que queremos obtener en nuestra petición.

En primer lugar, obtenemos las etiquetas de colores de todas nuestras imágenes llamando a la función "Get_Color_List" y realizamos el Kmeans de todas las imágenes con la mejor K para cada imagen, obtenida de findBestK.

A continuación, llamamos a la función "Retrieval_by_color", que recibe 3 parámetros:

- Las etiquetas de color obtenidas.
- La petición.
- El número de imágenes que queremos obtener.

Tras recorrer todas las etiquetas de color y comprobar si el color de la petición está en la etiqueta, guardando el índice en una lista en caso afirmativo, hacemos uso de "random.sample()" para obtener unos índices aleatorios que "teóricamente" cumplen la petición realizada (ya que el estudio nos demuestra que el algoritmo puede fallar).

Por último, verificamos para cada imagen que el color/es de la petición corresponda al que nos indica el GroundTruth.

De esta manera, en caso afirmativo, añadiremos un “True”, o en caso contrario añadiremos un “False”, creando así, una lista de booleanos que pasaremos por parámetro a la función “visualize_retrival” para que nos indique cuales son los aciertos y los fallos de nuestra petición.

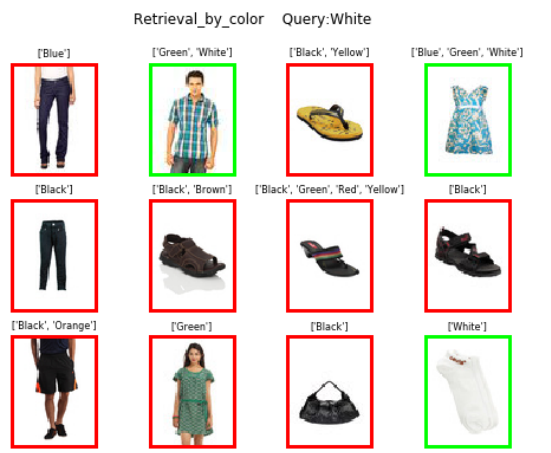
Hemos hecho dos tipos de pruebas:

- Analizando todos los colores con k=3, para que el algoritmo nos saque los 3 colores más relevantes.
- Analizando todos los colores con maxK=10, para que el algoritmo decida cuál es la k que mejor va a representar cada imagen.

A continuación, mostramos unos ejemplos de ejecución:

Debajo de las fotos, hemos mostrado la lista de colores que nos devuelve nuestro algoritmo kmeans cuando realiza la búsqueda.

<u>Test para k = 3</u>



[['White', 'Orange', 'Blue'], ['White', 'Blue', 'Orange'], ['White', 'Yellow', 'Black'], ['White', 'Blue', 'Blue'], ['White', 'Grey', 'Black'], ['White', 'Black', 'White'], ['White', 'Black', 'Red'], ['White', 'Black', 'Grey'], ['White', 'Red', 'Black'], ['White', 'Green', 'Orange'], ['White', 'Grey', 'Black'], ['White', 'White', 'Orange']]

[['White', 'Orange', 'Brown'], ['White', 'Orange', 'Grey'], ['White', 'White', 'Orange'], ['White', 'Orange', 'Purple'], ['White', 'Orange', 'Red'], ['White', 'White', 'Orange'], ['White', 'Orange', 'Red'], ['White', 'Orange', 'Blue'], ['White', 'Pink', 'Orange'], ['White', 'Black', 'Orange'], ['White', 'Orange', 'Black'], ['White', 'Black', 'Orange']]

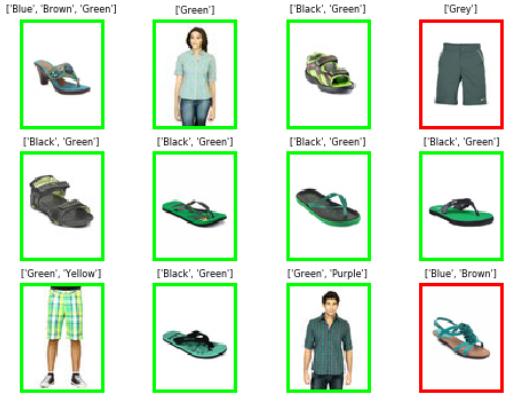
[['White', 'Grey', 'Black'], ['White', 'Grey', 'Blue'], ['White', 'Black', 'Grey'], ['White', 'Blue', 'Grey'], ['White', 'Grey', 'Black'], ['White', 'Grey', 'Black'], ['White', 'Grey', 'White'], ['White', 'Grey', 'Black'], ['White', 'Grey', 'Black'], ['White', 'Grey', 'Grey'], ['White', 'Black', 'Grey'], ['White', 'Grey', 'Black']]

[['White', 'Black', 'Blue'], ['White', 'Grey', 'Blue'], ['White', 'Blue', 'Blue'], ['Grey', 'White', 'Blue'], ['White', 'Blue', 'Purple'], ['White', 'Pink', 'Blue'], ['White', 'Grey', 'Blue'], ['White', 'Blue', 'Blue'], ['White', 'Blue', 'Blue'], ['White', 'Black', 'Blue'], ['White', 'Orange', 'Blue'], ['White', 'Blue', 'Grey']]

Retrieval_by_color Query:Brown


```
[['White', 'Yellow', 'Brown'], ['White', 'Orange', 'Brown'],
['White', 'Black', 'Brown'], ['White', 'Yellow', 'Brown'],
['White', 'Brown', 'Orange'], ['White', 'Brown', 'Grey'],
['White', 'Brown', 'Black'], ['White', 'Yellow', 'Brown'],
['White', 'Brown', 'Yellow'], ['White', 'Orange', 'Brown'],
['White', 'Brown', 'Orange'], ['White', 'Brown', 'Orange']]
```

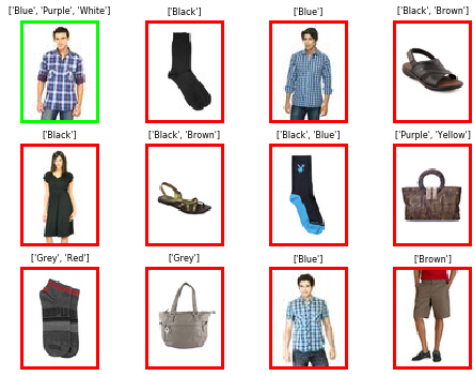
Retrieval_by_color Query:Green




```
[['White', 'Green', 'Grey'], ['White', 'Green', 'Grey'], ['White',
'Green', 'Green'], ['White', 'Grey', 'Green'], ['White', 'Green',
'Grey'], ['White', 'Green', 'Green'], ['White', 'Green', 'Green'],
['White', 'Green', 'Green'], ['White', 'Brown', 'Green'],
['White', 'Green', 'Green'], ['White', 'Orange', 'Green'],
['Grey', 'White', 'Green']]
```

Test para k = 10

Retrieval_by_color Query:White

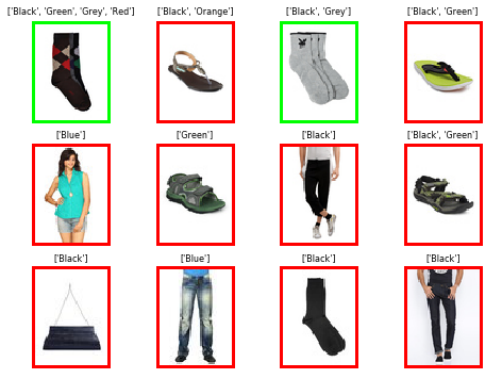



```
[['White', 'Blue', 'Blue', 'Orange', 'Blue', 'Blue'], ['White',
'Black', 'White', 'Grey', 'Grey'], ['White', 'Blue', 'Orange',
'Blue', 'White', 'Blue'], ['White', 'Grey', 'Red', 'Grey', 'White',
'Black'], ['White', 'Orange', 'Orange', 'Orange', 'Brown',
'Black'], ['White', 'White', 'Yellow', 'Brown', 'Yellow', 'Black'],
['White', 'Blue', 'Blue'], ['White', 'Grey', 'Pink', 'Brown',
'Red'], ['White', 'Grey', 'Black', 'Red'], ['White', 'Grey',
'White', 'Grey', 'Grey', 'Grey'], ['White', 'Blue', 'White',
'Blue', 'Grey'], ['White', 'Grey', 'Brown', 'Brown', 'Orange',
'Red']]
```

Retrieval_by_color Query:Orange


```
[['White', 'Grey', 'Orange', 'Blue'], ['White', 'Blue', 'Black',
'Orange'], ['White', 'Brown', 'White', 'Orange', 'Black'],
['White', 'Red', 'Blue', 'Orange', 'Red', 'Pink'], ['White',
'Orange', 'Pink', 'Black', 'Purple', 'Purple'], ['White', 'Purple',
'Pink', 'Red', 'Orange', 'Purple'], ['White', 'Orange', 'Orange',
'Black', 'Orange', 'Brown'], ['White', 'Orange', 'Orange', 'White',
'Brown', 'Black'], ['White', 'Purple', 'Orange', 'Orange', 'Blue'],
['White', 'Blue', 'Orange', 'Blue', 'White', 'Blue'], ['White',
'Orange', 'Brown', 'Orange'], ['White', 'Brown', 'Orange', 'Brown',
'Orange', 'Red']]
```

Retrieval_by_color Query:Grey




```
[['White', 'Red', 'Orange', 'Grey', 'Black'], ['White', 'Orange',
'White', 'Grey', 'Orange', 'Black'], ['White', 'White', 'White',
'Grey', 'Grey', 'Grey', 'Grey', 'Grey'], ['White', 'Green', 'Grey',
'Green'], ['White', 'Orange', 'Green', 'Grey', 'Orange', 'Green',
'Brown'], ['White', 'Green', 'Grey', 'Green', 'Green', 'White'],
['White', 'Orange', 'White', 'Orange', 'Grey', 'Black'], ['White',
'Grey', 'Green', 'Grey', 'Black'], ['White', 'White', 'Blue',
'Blue', 'Grey'], ['White', 'Grey', 'White', 'Grey', 'Blue',
'Black', 'Blue', 'Blue'], ['White', 'Black', 'White', 'Grey',
'Grey'], ['White', 'Grey', 'Orange', 'Grey', 'Black', 'White']]
```

Retrieval_by_color Query:Blue


```
[['White', 'Blue', 'Blue', 'Blue', 'White', 'Blue'], ['White',
'Grey', 'White', 'Orange', 'Blue', 'Brown'], ['White', 'Blue',
'Blue', 'Blue'], ['Orange', 'White', 'Blue', 'Black'], ['White',
'Grey', 'Grey', 'Orange', 'Black', 'Blue'], ['White', 'Blue',
'Orange', 'Blue', 'Blue', 'Orange', 'Black'], ['White', 'Orange',
'Blue'], ['White', 'Blue', 'Orange', 'Orange', 'Orange', 'Red',
'Brown'], ['Blue', 'Blue', 'Orange', 'White', 'Blue', 'Blue'],
['White', 'Blue', 'Grey', 'Blue', 'Grey'], ['White', 'Grey',
'Grey', 'Blue', 'Pink'], ['Blue', 'Blue', 'Orange', 'Brown',
'White']]
```

Podemos observar algunos errores en el algoritmo para k=3, esto puede ser debido a que unos colores están más presentes que otros o a una confusión por las tonalidades.

El naranja lo encontramos en la piel de las personas, y puesto que estamos analizando prendas de ropa con modelos para mostrárnosla, la confusión es muy elevada. Además, hay ciertas tonalidades que pueden dar a confusión con el amarillo, gris clarito e incluso marrón.

El gris, no es un ejemplo claro de abundante presencia en objetos, pero sí que las tonalidades son mezclas de diferentes tonos de blancos y negros, e incluso se puede dar lugar a confusión con algún azul. Esto hace por lo tanto que sea otro color con el cual nuestro algoritmo falla mucho más.

El color blanco, de hecho, nuestro algoritmo lo asigna a todas las imágenes por culpa del fondo blanco, haciendo que sea muy difícil acertar.

Para maxK = 10, observamos que nuestro etiquetador falla un poco más en general, esta ligera decaída se debe a que cuando trabajamos con una k más pequeña, el algoritmo escogerá los colores más probables en la foto, por lo tanto, se acercará a escoger los predominantes de la imagen.

Un problema que observamos es la dificultad de determinar si una prenda es de color blanco, ya que todas las imágenes tienen un fondo blanco que confunde al algoritmo. Una posible solución sería calcular cual es el porcentaje medio de blanco que tenemos en nuestras imágenes, ya que como todas siguen un mismo patrón de fondo blanco podríamos llegar a aproximar el porcentaje del valor medio y el cual fijaríamos como una constante en nuestro etiquetador para diferenciar entre el blanco de las prendas y del fondo.

Otro problema es el hecho de que los modelos afectan a la respuesta del algoritmo, por lo que se podrían separar las imágenes en 2 grupos, las que tienen modelos humanos y las que no. La finalidad de esta separación es hacer con el color naranja lo mismo que hemos hecho con el blanco, de esta manera tendríamos un porcentaje medio de naranja por foto que sería constante, y junto con el porcentaje asignado a cada color, podríamos llegar a ser un poco más precisos con nuestras búsquedas.

Como conclusión del estudio, notamos que no es un etiquetador preciso, y que no sería apto para lanzarlo al mercado directamente. Seria necesario aplicar ciertas mejoras como las comentadas anteriormente. Estas mejoras podrían reducir mínimamente el acierto en algunos colores, pero consideramos que el hecho de tener un etiquetador más estable y equilibrado es beneficioso.

Test_Retrieval_by_shape

Este test nos permite saber si nuestro algoritmo KNN acierta con las etiquetas de forma que nos devuelve, comparándolas con el GroundTruth que se nos proporciona.

La función recibe 4 parámetros:

- Una lista con todas las imágenes a evaluar.
- El número de vecinos (labels) que queremos que mire nuestro algoritmo KNN.
- La petición realizada, en este caso, una prenda (string).
- El número de imágenes que queremos obtener en nuestra petición.

En primer lugar, inicializamos nuestro KNN con el entrenamiento para acto seguido aplicar la función "predict" que es lo que nos clasifica las imágenes (es en esta donde le indicamos cuantos vecinos(labels) queremos que mire). Una vez obtenemos el resultado de todas las imágenes clasificadas llamamos a la función "Retrieval_by_shape" que recibe 3 parámetros:

- El resultado de aplicar el knn con nuestras imágenes (etiquetas de forma).
- La petición.
- El número de imágenes que queremos obtener.

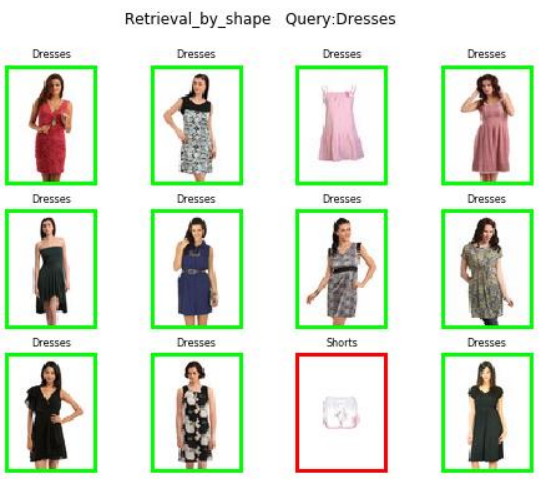
De igual forma que en el retrieval_by_color, tras recorrer todas las etiquetas de forma y comprobar si la forma de la petición está en la etiqueta, guardando el índice en una lista en caso afirmativo, hacemos uso de "random.sample()" para obtener unos índices aleatorios que "teóricamente" cumplen la petición realizada.

Por último, y también usando la misma técnica que en el Test_Retrieval_by_color, verificar para cada índice de nuestras imágenes, usando el GroundTruth, que la forma de la petición este en el GroundTruth correspondiente al índice de nuestra imagen evaluada.

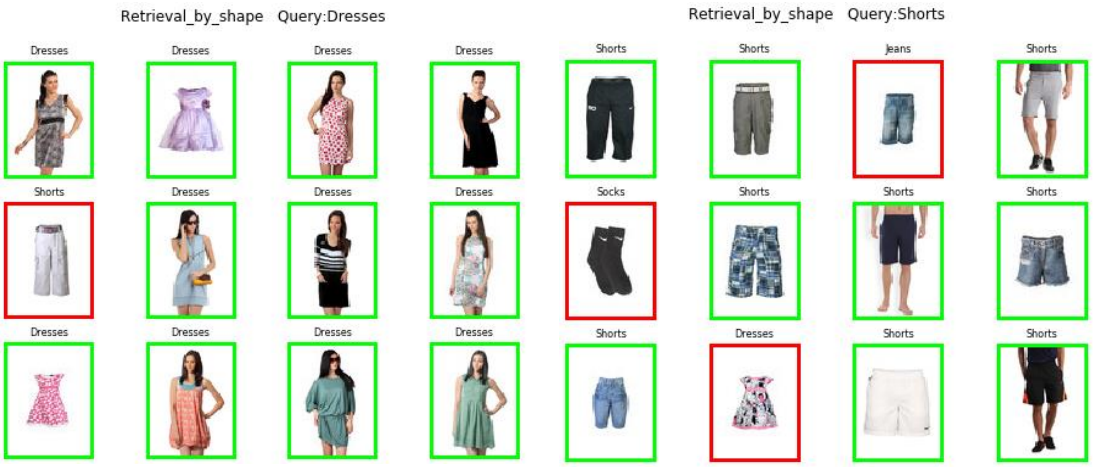
De esta manera, en caso afirmativo, añadiremos un "True", o en caso contrario añadiremos un "False", creando así, una lista de booleanos que pasaremos por parámetro a la función "visualize_retrival" para que nos indique cuales son los aciertos y los fallos de nuestra petición.

A continuación, mostramos los diferentes experimentos realizados:

Test para k = 3



Test para k = 10



Observamos a diferencia del Kmeans que el algoritmo KNN es mucho más estable y el porcentaje de acierto es mucho más elevado. Notamos que el hecho de variar el valor de K no varía mucho la calidad de nuestro etiquetador basado en la forma de las prendas de ropa. Esto sucede para valores bajos.

Si cogemos valores muy grandes sí que sería del todo perjudicial, ya que el algoritmo en el momento de buscar los vecinos cercanos acabaría seleccionando vecinos de clases erróneas.

Ahora bien, en los niveles en los que estamos trabajando con nuestro algoritmo KNN, podemos observar que las diferencias cualitativas son ínfimas.

Un ejemplo donde se ve claramente que a nuestro algoritmo le puede costar más es cuando buscamos “Flip Flops”, ya que las “Sandals” i las “Heels” son también calzado, y además a pie descubierto.

Test_Visualize_Kmeans

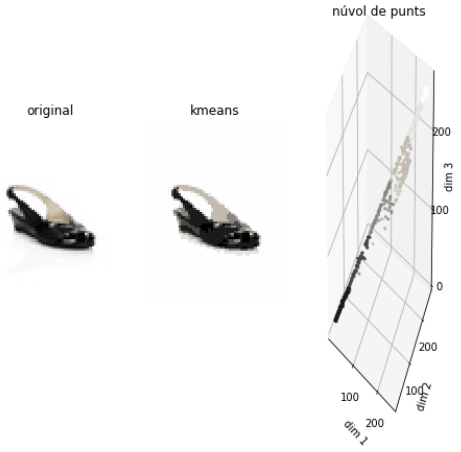
Este test nos permite apreciar la calidad de nuestro algoritmo Kmeans y como representa la imagen en función de la k que el algoritmo otorgue como mejor.

La k escogida es la necesaria para representar los colores principales de la foto y por lo tanto poderla representar correctamente.

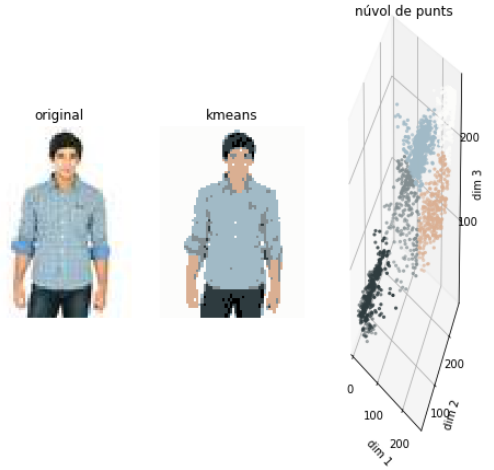
Para realizar este test, como ahora simplemente estamos valorando la calidad de visualización, escogemos los índices al azar al principio. Después aplicamos el Kmeans para cada imagen y le pedimos que nos devuelva los colores encontrados y también la k escogida para visualizar la imagen.

A continuación, mostramos algunos ejemplos:

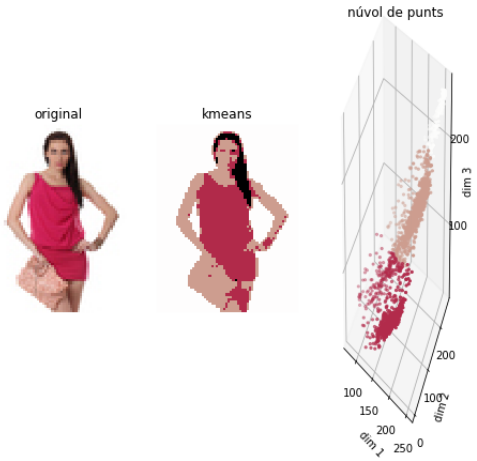
--------Imatge representada amb K=6----------
Colors trobats pel kmeans:['White', 'Black', 'White', 'Grey', 'Grey', 'Grey']



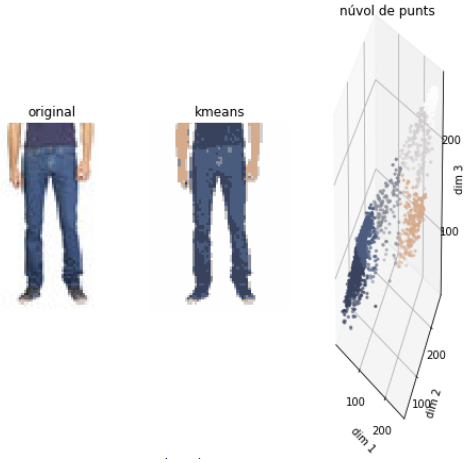
--------Imatge representada amb K=5----------
Colors trobats pel kmeans:['White', 'Blue', 'Grey', 'Orange', 'Black']



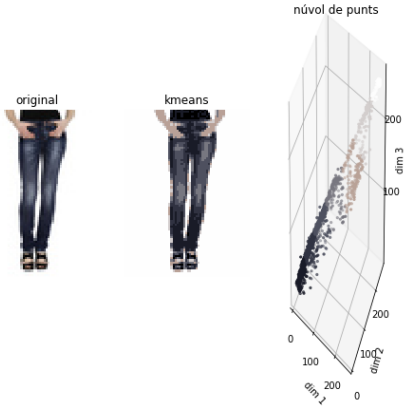
--------Imatge representada amb K=3----------
Colors trobats pel kmeans:['White', 'Red', 'Orange']



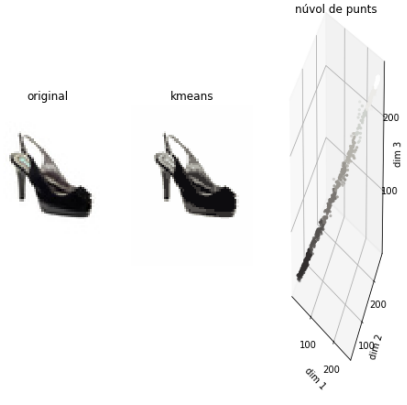
--------Imatge representada amb K=6----------
Colors trobats pel kmeans:['White', 'Blue', 'Blue', 'White', 'Orange', 'Blue']



--------Imatge representada amb K=7----------
Colors trobats pel kmeans:['White', 'White', 'Blue', 'Orange', 'Blue', 'Blue', 'Grey']



--------Imatge representada amb K=8----------
Colors trobats pel kmeans:['White', 'Grey', 'White', 'Grey', 'Grey', 'Black', 'White', 'Grey']



La conclusión que podemos extraer después de este test es que las imágenes con más detalles, y más colores destacados o con diferentes tonalidades, tienden a ser representadas con una k más elevada, sin embargo, las imágenes más sencillas y con menos matices tienden a conformarse con un valor de k inferior. También podemos observar que las representaciones con las etiquetas escogidas por el algoritmo son bastante buenas, ya que se sigue pudiendo reconocer la prenda y los colores.

Dos claros ejemplos son: los Heels (k = 8) y el Dress (k = 3).

Análisis cuantitativo:

En el análisis cuantitativo de Kmeans y KNN que se realizará a continuación estudiaremos los porcentajes de éxito y tiempo consumido por ambos algoritmos y su evolución cuando el número de centroides o vecinos es modificado. Se han programados dos funciones por algoritmo, un test que realizará todos los preparativos del estudio y una función que analiza los resultados obtenidos.

Test forma

Empezaremos realizando el estudio de KNN, para el cual se han programado lo siguiente:

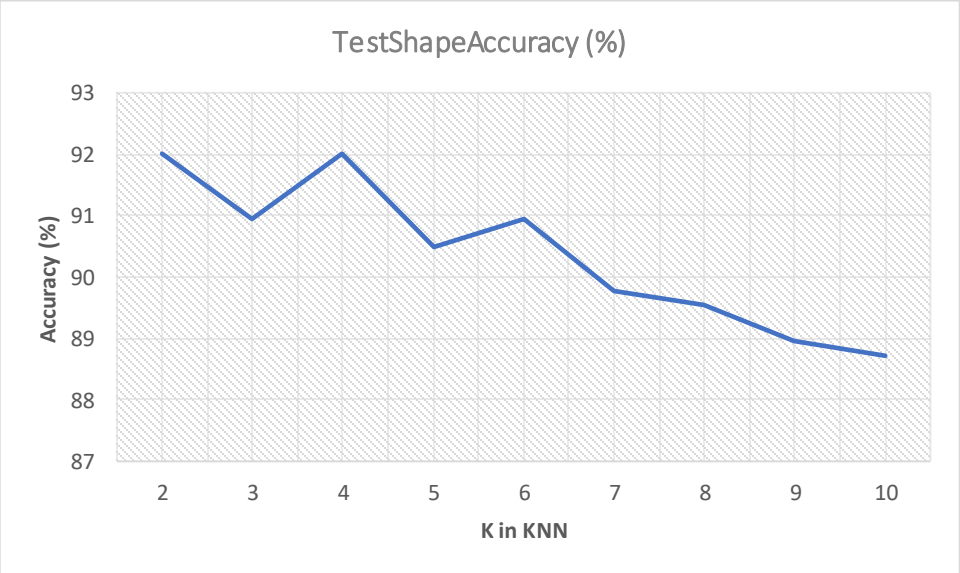
- **TestShapeAccuray:**
  Se encargará inicializar la variable KNN con el conjunto de entrenamiento, que si se requiere habrá sido reducido, y obtener las predicciones de forma que proporciona el algoritmo. Con estas predicciones llamará a la función Get_shape_accuracy para obtener el porcentaje de acierto respecto al Ground-Truth y devolverá este porcentaje.

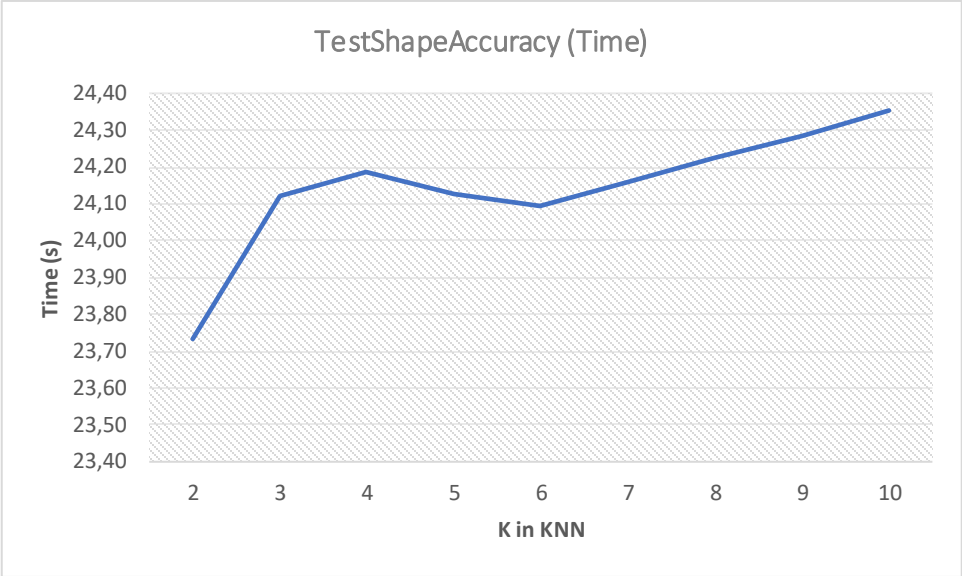
  Recibe como parámetro de entrada las imágenes de entrenamiento y sus etiquetas, las imágenes de test y sus etiquetas (Ground-Truth), el número de vecinos que queremos tener en cuenta y el porcentaje de imágenes de entrenamiento que queremos usar, este último parámetro y su efecto en el algoritmo lo estudiaremos en el apartado de modificaciones.

- **Get_shape_accuracy:**
  Recibe como parámetro de entrada las etiquetas obtenidas al aplicar el KNN y el Ground-Truth y se compararan ambos parámetros para obtener el porcentaje de acierto.

Con las siguientes funciones aplicadas sobre el programa base podemos ver el comportamiento del algoritmo a medida que se modifica la cantidad de vecinos a tener en cuenta. Las siguientes dos gráficas se han hecho desde 2 hasta 10 vecinos y muestran el porcentaje de error y tiempo transcurrido:

	2	3	4	5	6	7	8	9	10
Accuracy (%)	92,01	90,95	92,01	90,48	90,95	89,78	89,54	88,95	88,72
Time (s)	23,73	24,12	24,19	24,13	24,10	24,16	24,22	24,28	24,35





Uno de los inconvenientes del KNN es tener que escoger una buena K para obtener los mejores resultados, aumentar o disminuir este valor no siempre tendrá las mismas repercusiones. Podemos ver este inconveniente reflejado en la primera gráfica, donde aumentar de 2 a 3 vecinos reduce la tasa de acierto un 1% pero aumentarlos de 3 a 4 aumenta la tasa también en un 1%. Podemos observar que la tasa de acierto para el rango de valores escogidos no varía mucho (un 3,3% de diferencia máxima) pero si siguiésemos aumentando la cantidad de vecinos la tasa de aciertos caería en picado para valores muy grandes, ya que se tendrían en cuenta muchos vecinos que no corresponderán con la clase correcta. Por ejemplo, en el caso de K=100 la tasa de aciertos es del 43,36%.

En cuanto al rendimiento respecto al tiempo, observamos que las variaciones en la cantidad de vecinos afectan poco al tiempo consumido por el algoritmo, solo se llega a observar una variación máxima ligeramente superior al medio segundo respecto a los 24-23 que tarda en ejecutarse el algoritmo. Para complementar el caso concreto dicho anteriormente, para K=100 el algoritmo tarda 24,26 segundos.

Con este análisis podemos concluir que el valor de vecinos que se tienen en cuenta no afecta prácticamente nada en el tiempo que tarda en ejecutarse el programa, por lo que si tuviéramos que decidir la cantidad de vecinos que se tienen en cuenta recae únicamente deberíamos tener en consideración la tasa de acierto.

Test color

Para el análisis de Kmeans se han programado funciones equivalentes a las usadas para KNN, estas son:

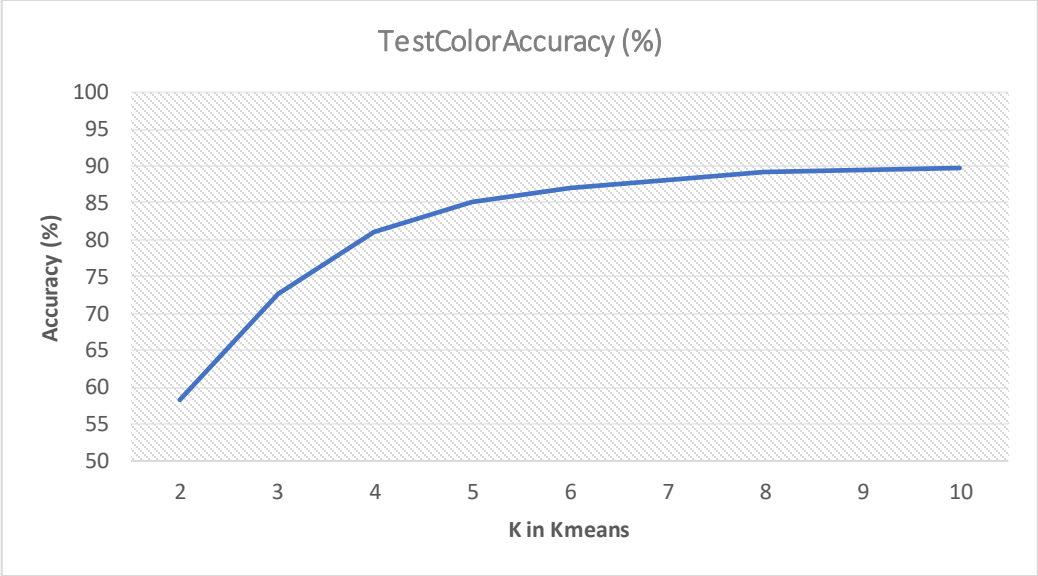
- **TestColorAccuray:**
  Se encargará de ejecutar el algoritmo Kmeans para cada imagen por separado, ya que con este algoritmo solo podemos hacer las predicciones de color de una imagen a la vez. Todas las predicciones serán almacenadas para poder calcular la tasa de acierto llamando a la función Get_color_Accuracy.

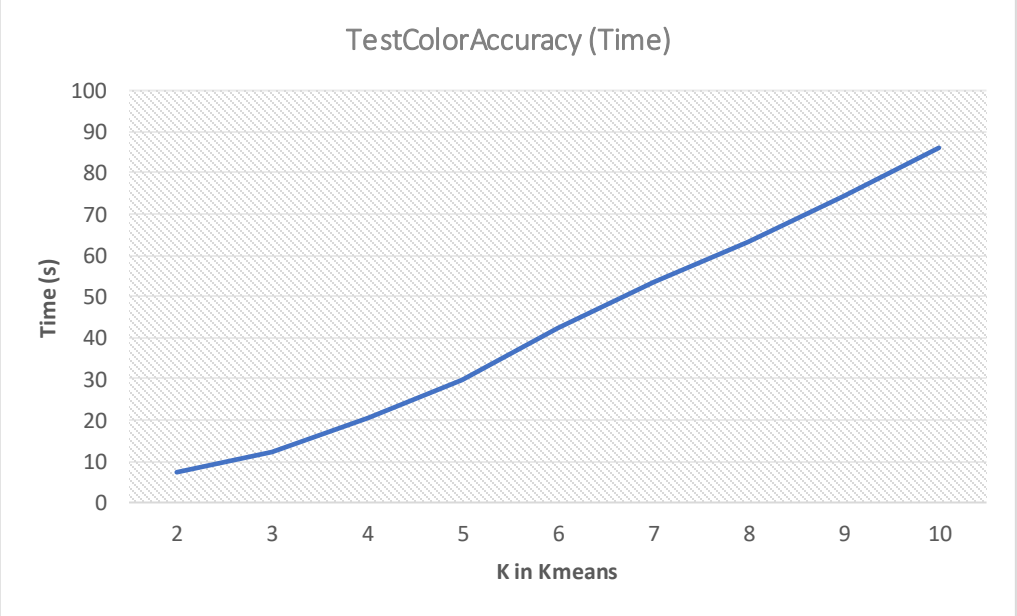
Recibe como parámetro de entrada las imágenes de test y sus etiquetas (Ground-Truth), el número de centroides y las opciones de Kmeans, este último parámetro y su efecto en el algoritmo lo estudiaremos en el apartado de modificaciones.

- **Get_color_accuracy:**
  Recibe como parámetro de entrada los colores obtenidos para cada imagen y el Ground-Truth, se utilizan estos valores para determinar la tasa de acierto del algoritmo. El criterio utilizado para determinar esta tasa consiste en calcular el porcentaje de colores del Ground-Truth presentes en las predicciones.

Al igual que hemos hecho con KNN, aplicaremos las funciones sobre el programa base para obtener las métricas de tasa de acierto y tiempo trascurrido:

	2	3	4	5	6	7	8	9	10
Accuracy (%)	58,27	72,65	81,01	85,00	87,12	88,16	89,18	89,38	89,74
Time (s)	7,34	12,49	20,40	29,88	42,24	53,63	63,32	74,38	86,10





Dado el criterio de acierto actual para el algoritmo de Kmeans al aumentar la cantidad de centroides, que en este caso representan los colores, la tasa de acierto siempre tiende a aumentar. Como más posibles colores ofrezcamos al algoritmo más probabilidad tiene de poder acertar todos o algunos de los colores de cada imagen. Sabiendo esto, lo normal sería siempre utilizar un número elevado de K para obtener una tasa de acierto elevada, pero elevar el valor de K afecta gravemente al rendimiento del programa y se deberá justificar el aumentar este valor.

El rendimiento de este algoritmo está directamente ligado al valor de K ya que como más grande sea este valor más operaciones son necesarias de ejecutar y más se tardará en convergir. Dada esta situación se entra en un compromiso entre tiempo y tasa de acierto donde se tendrá que decidir qué propiedad priorizar y cuanto de esta se puede sacrificar para obtener mejores valores de la otra. Por ejemplo, si lo que buscamos es la máxima precisión posible dentro de los rangos de K mostrados podríamos escoger K=8 por encima de K=10, ya que sacrificamos un 0,56% de los aciertos a cambio de ejecutar el algoritmo más de 20 segundos más rápido.

Otra opción distinta a fijar un valor de K sería utilizar el algoritmo de findBestK para asignar la mejor cantidad de centroides a cada imagen. Esto sería conveniente en casos donde trabajamos con un conjunto reducido de imágenes y queremos los mejores resultados, pero en situaciones donde tengamos que analizar grandes cantidades de imágenes puede ser contraproducente en términos de tiempo, ya que este algoritmo aplica el Kmeans repetidas veces en cada imagen y aumenta el tiempo de ejecución necesario.

Cabe destacar que el criterio de acierto utilizado puede variar mucho depende el uso que se le quiera dar al algoritmo y, por lo tanto, la gráfica mostrada no es representativa para todos los casos de este algoritmo.

Modificaciones/Mejoras:

En este apartado propondremos distintas mejoras para ambos algoritmos, centrándonos más en el análisis del comportamiento de KNN y los cambios de comportamiento con los cambios propuestos. El motivo de centrarnos más en el compartimiento del KNN radica en que el criterio de acierto es siempre el mismo, se ha acertado o no respecto al Ground-Truth y resulta más interesante ver estos resultados que los del Kmeans, donde la definición del criterio es variable a cada aplicación.

Efecto de las propiedades de las imágenes en KNN

El primer conjunto de cambios reside en la modificación de las imágenes utilizadas, donde estas serán cambiadas de tamaño y resolución. Para realizar estas modificaciones hemos usado las siguientes páginas web, ya que permitían la transformación simultanea de una gran cantidad de imágenes: https://bulkresizephotos.com/ y https://www.birme.net/.

A continuación, se puede observar la imagen original y las 3 variaciones realizadas de esta:


Original (60x80)


Baja resolución (60x80)


Reducida 50% (30x40)

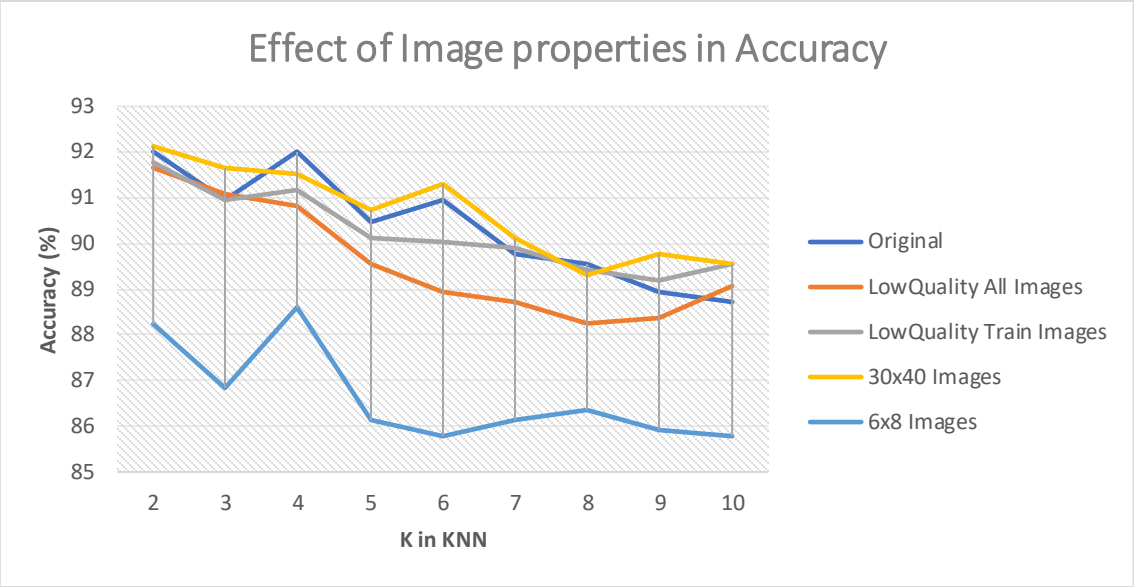

Reducida 90% (6x8)

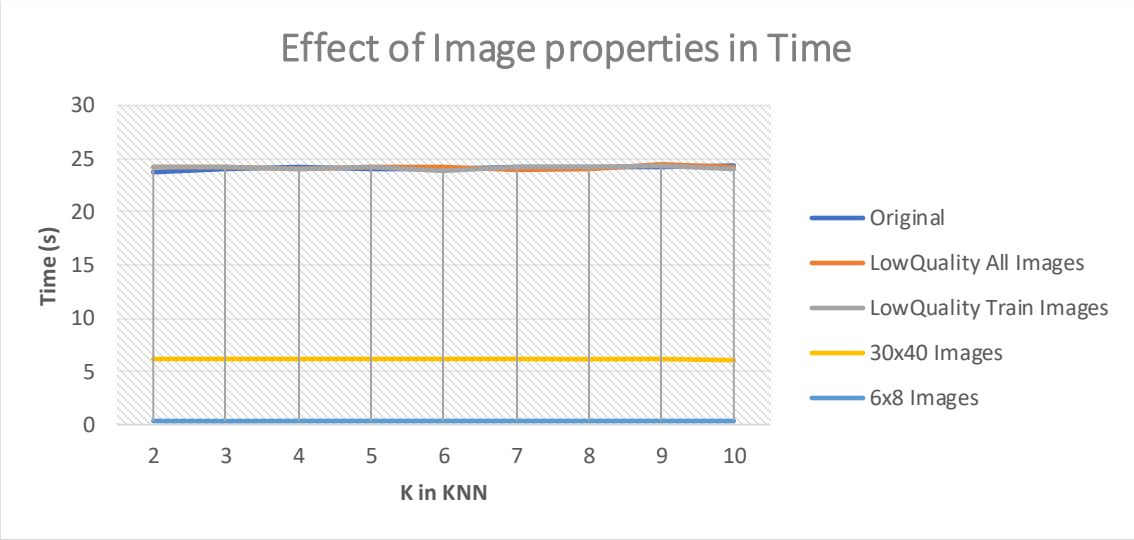
La finalidad de estas modificaciones es observar si el comportamiento del programa es aceptable en estas condiciones o si, en el mejor de los casos, se comporta de una mejor forma, ya que si decidimos utilizar estas imágenes y no las originales el espacio reservado en memoria será muy inferior (Train set original = 46MB, Train set 30x40 = 2,30MB).

Los experimentos realizados con estos cambios consistirán en los mismos hechos anteriormente con las imágenes originales, pero con todas las imágenes modificadas. En el caso de la modificación de baja resolución se realizarán 2 experimentos, uno con todas las imágenes a baja resolución y otro con solo las imágenes del set de entrenamiento modificadas. A continuación, se presentan los resultados obtenidos:

TestShapeAccuracy (%)	2	3	4	5	6	7	8	9	10
Original	92,01	90,95	92,01	90,48	90,95	89,78	89,54	88,95	88,72
LowQuality All Images	91,66	91,07	90,83	89,54	88,95	88,72	88,25	88,37	89,07
LowQuality Train Images	91,77	90,95	91,19	90,13	90,01	89,89	89,42	89,19	89,54
30x40 Images	92,13	91,66	91,54	90,72	91,30	90,13	89,31	89,78	89,54
6x8 Images	88,25	86,84	88,60	86,13	85,78	86,13	86,37	85,90	85,78

TestShapeAccuracy (Time)	2	3	4	5	6	7	8	9	10
Original	23,73	24,12	24,19	24,13	24,10	24,16	24,22	24,28	24,35
LowQuality All Images	24,16	24,17	24,09	24,18	24,17	23,95	24,04	24,47	24,14
LowQuality Train Images	24,14	24,18	24,04	24,25	23,86	24,26	24,15	24,33	24,11
30x40 Images	6,12	6,11	6,14	6,13	6,14	6,13	6,16	6,12	6,06
6x8 Images	0,34	0,34	0,34	0,35	0,34	0,34	0,34	0,34	0,34





Al reducir la calidad de las imágenes estas se ven peor, pero la forma de las prendas se mantiene prácticamente igual y eso es lo que nos interesa a la hora de utilizar el KNN. Como podemos observar en los resultados, en los dos experimentos llevados a cabo con las imágenes de baja calidad la respuesta ha sido muy positiva con una tasa de éxito muy cercana a la obtenida en el experimento original. Teniendo esto en cuenta, a la hora de utilizar el algoritmo en una aplicación real podríamos considerar la opción de reducir la calidad de las imágenes para reducir la memoria requerida si es posible permitirse la ligera reducción de la tasa de éxito.

En cuanto el tiempo transcurrido no hay ninguna mejora, el motivo de esto es que el algoritmo sigue realizando la misma cantidad de trabajo ya que las imágenes siguen teniendo la misma cantidad de puntos. Para solucionar este problema y buscar la reducción del tiempo del algoritmo, y el ahorro de memoria para almacenar imágenes, se ha propuesto la reducción de dimensión de las imágenes ya que contendrán menos puntos.

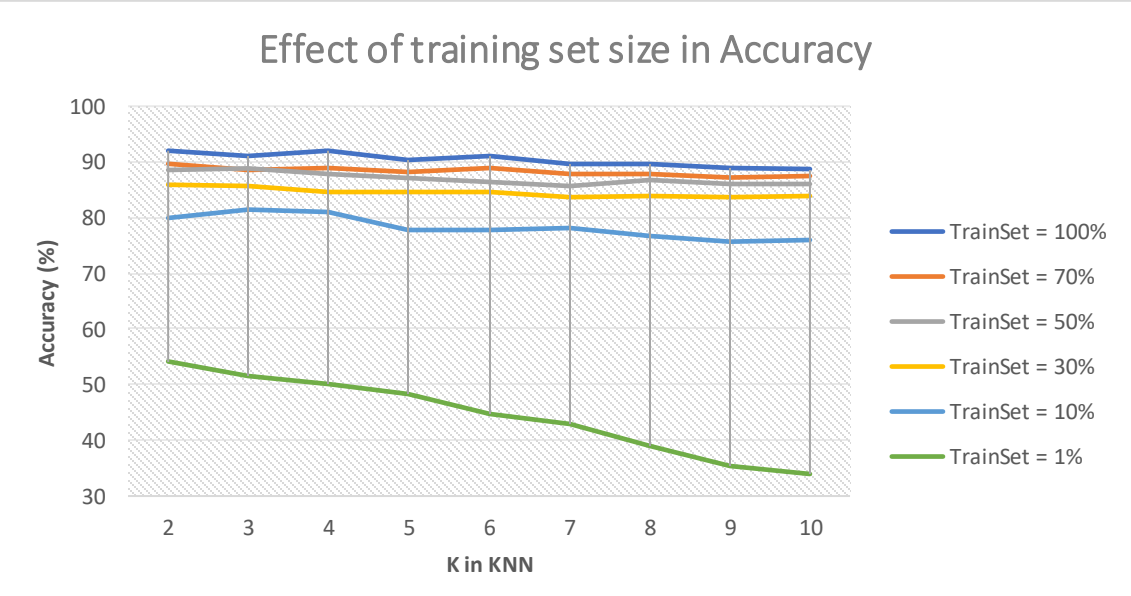
Como suponíamos, el tiempo consumido por el algoritmo con las imágenes reducidas es muy inferior, llegando al extremo de tardar prácticamente 0 segundos en el caso de imágenes de 6x8. El inconveniente de reducir las imágenes es que como más pequeñas sean más complicado es de discernir para el ser humano la forma de lo que está observando, pero esto no parece un inconveniente para el algoritmo, ya que su tasa de éxito para imágenes de 30x40 es prácticamente idéntico al que obteníamos con las imágenes originales e incluso, en algunos casos, superior. En el caso de 6x8, que a simple vista no somos capaces de decir de que prenda se trata, el algoritmo presenta resultados realmente excepcionales dada la situación. En este caso la tasa de acierto es inferior, pero solo hay una diferencia de acierto del 4% entre este experimento y el realizado con las imágenes originales.

Con estos resultados podemos estar seguros de que reducir el tamaño de las imágenes sería algo casi imprescindible en muchos casos, ya que el tiempo consumido por el algoritmo baja drásticamente, la tasa de éxito es similar y, además, las imágenes reducidas también ocupan menos espacio en memoria. La cantidad de reducción que se quiera aplicar habría que estudiarla en cada aplicación de uso, de esta forma decidir cuanta tasa de éxito se está dispuesto a perder (si la reducción es muy grande) a cambio de todas las ventajas que ofrece esta modificación.

Efecto de la longitud del set de entrenamiento

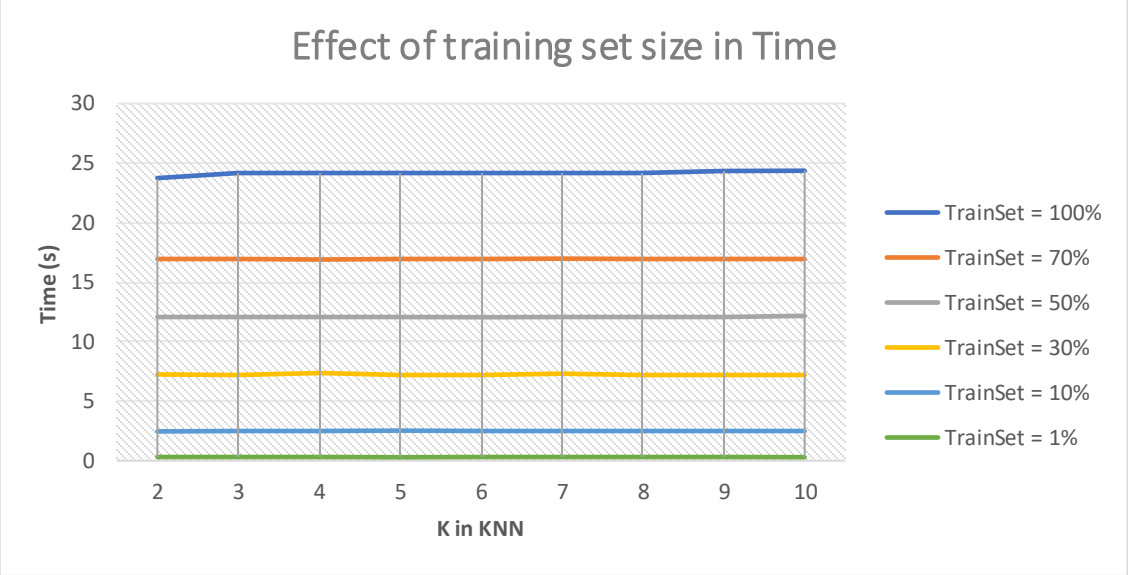
El siguiente cambio que realizaremos será la modificación de la longitud del set de entrenamiento, realizaremos los experimentos sobre el programa base para diferentes longitudes y observaremos que sucede cuando esta se va reduciendo. La motivación de este cambio reside en el hecho de que el rendimiento del algoritmo está directamente ligado al set de entrenamiento, ya que como menos imágenes tenga menos operaciones realizará el algoritmo al buscar los K vecinos más próximos.

TestShapeAccuracy (%)	2	3	4	5	6	7	8	9	10
TrainSet = 100%	92,01	90,95	92,01	90,48	90,95	89,78	89,54	88,95	88,72
TrainSet = 70%	89,66	88,48	88,84	88,25	88,84	87,90	87,66	87,19	87,31
TrainSet = 50%	88,60	88,84	87,66	87,19	86,37	85,66	86,60	85,90	85,90
TrainSet = 30%	85,90	85,55	84,72	84,72	84,49	83,67	84,02	83,67	83,78
TrainSet = 10%	80,02	81,43	80,96	77,79	77,91	78,03	76,85	75,68	75,91
TrainSet = 1%	54,17	51,47	49,94	48,18	44,89	43,01	39,13	35,25	33,96



El comportamiento obtenido es el esperado, ya que a medida que reducimos el set de entrenamiento menos referencias de cada prenda hay, por lo que a la hora de decidir a qué prenda corresponde a cada imagen la probabilidad de cometer un error es mayor. De la misma forma, si aumentamos el set también aumentaremos el porcentaje de éxito. Observamos que la reducción de tasa de éxito es bastante aceptable en las reducciones del set a un 70% y 50%, pero que en el caso de reducirla a 1% la tasa de éxito es poco fiable. En los otros casos la diferencia en la tasa de acierto habría que considerar si son aceptables para el uso que se le da al algoritmo.

TestShapeAccuracy (Time)	2	3	4	5	6	7	8	9	10
TrainSet = 100%	23,73	24,12	24,19	24,13	24,10	24,16	24,22	24,28	24,35
TrainSet = 70%	16,95	16,93	16,90	16,93	16,91	16,97	16,92	16,96	16,90
TrainSet = 50%	12,15	12,13	12,13	12,08	12,05	12,09	12,11	12,09	12,16
TrainSet = 30%	7,25	7,27	7,30	7,25	7,27	7,31	7,25	7,27	7,27
TrainSet = 10%	2,46	2,53	2,47	2,53	2,47	2,46	2,48	2,48	2,48
TrainSet = 1%	0,29	0,29	0,29	0,29	0,29	0,29	0,29	0,29	0,29



Se puede observar una clara diferencia de tiempo de ejecución para los distintos tamaños de set, como ya habíamos explicado anteriormente, y que este se reduce en el mismo porcentaje en el que se reduce el set. En el caso de reducir al 50% el set de entrenamiento el tiempo también se reduce un 50%, esta información es muy valiosa ya que podremos estimar aproximadamente el tiempo que tomará el algoritmo y experimentar directamente con los valores que nos aseguren nuestros objetivos referentes al tiempo.

Si tuviéramos que escoger un porcentaje de set de entrenamiento para un propósito general del algoritmo escogeríamos un 50%, ya que el algoritmo tardaría la mitad de tiempo en ejecutarse a cambio de tener una tasa de éxito ligeramente inferior.

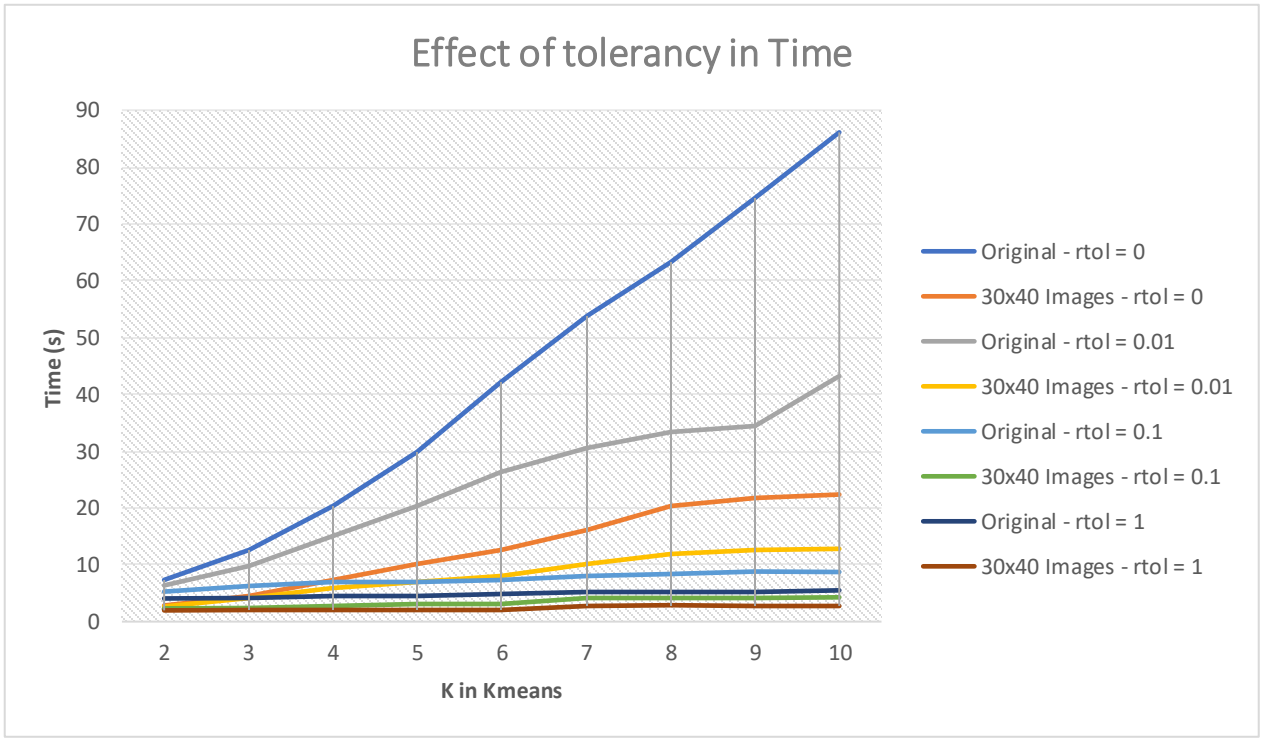
Efecto tolerancia y tamaño de imagen en Kmeans

Para finalizar este apartado experimentaremos cómo se comporta Kmeans si modificamos la tolerancia relativa a la hora de comparar los centroides actuales con los anteriores, es decir, a la hora de decidir si ha convergido. La tolerancia en nuestro código se aplica de la siguiente forma: $|OldCentroids - Centroids| \leq rtol * |Centroids|$

El programa base se ha estado ejecutando con una tolerancia relativa de 0, esto significa que solo se convergía si los centroides actuales y antiguos eran exactamente iguales. Al no tener tolerancia el programa puede tardar bastante en convergir, por lo que si añadimos un poco de tolerancia se espera que el tiempo de ejecución se reduzca.

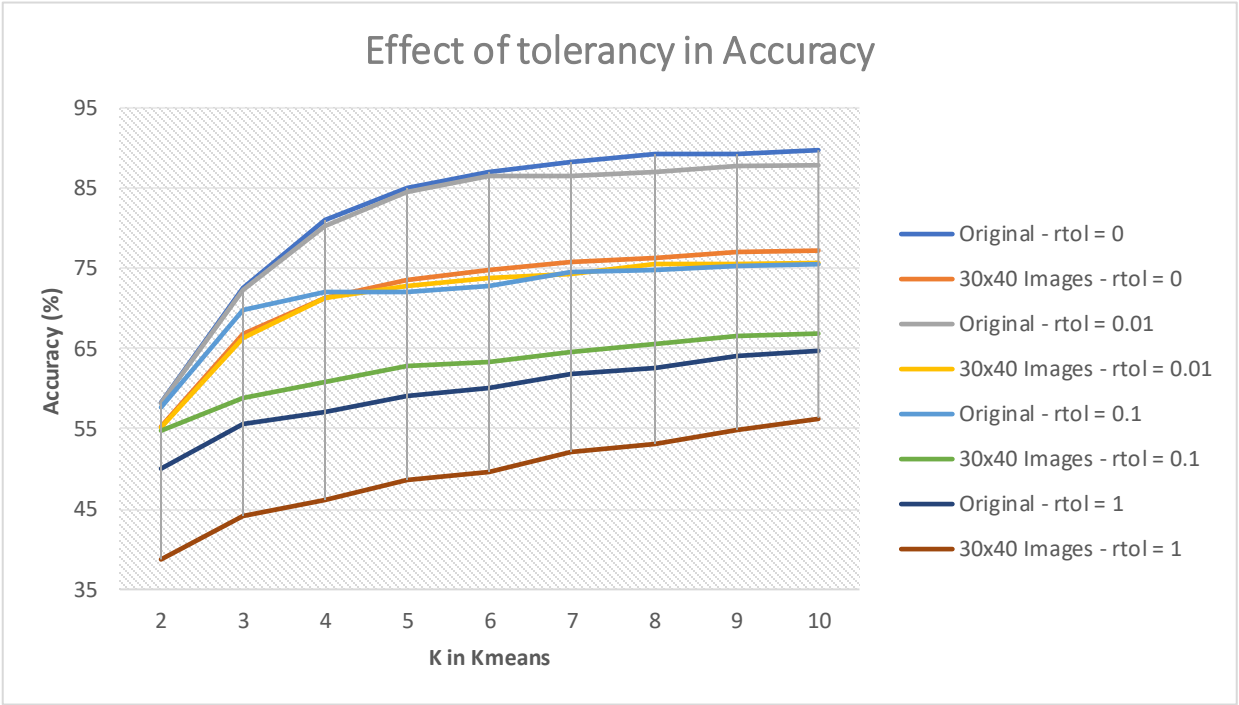
A continuación, podremos ver si es cierto que se reduce el tiempo y que efecto tiene la tolerancia en la tasa de aciertos. Este experimento también se va a realizar con imágenes reducidas al 50% para poder observar la capacidad de detección de color en una situación algo más complicada y el efecto en el rendimiento.

TestColorAccuracy (Time)	2	3	4	5	6	7	8	9	10
Original - rtol = 0	7,34	12,49	20,40	29,88	42,24	53,63	63,32	74,38	86,10
30x40 Images - rtol = 0	2,84	4,56	7,16	10,02	12,53	16,16	20,21	21,65	22,36
Original - rtol = 0.01	6,37	9,90	14,94	20,44	26,41	30,69	33,45	34,44	43,21
30x40 Images - rtol = 0.01	2,66	4,07	5,94	7,01	7,98	10,12	11,85	12,64	12,81
Original - rtol = 0.1	5,28	6,40	6,91	6,87	7,44	7,85	8,44	8,81	8,75
30x40 Images - rtol = 0.1	2,33	2,57	2,67	3,11	3,11	4,00	4,27	4,15	4,28
Original - rtol = 1	4,07	4,26	4,48	4,66	4,80	5,19	5,11	5,34	5,50
30x40 Images - rtol = 1	1,97	2,01	2,21	2,11	2,16	2,61	2,89	2,83	2,86



Con estos datos corroboramos la teoría anterior, simplemente con añadir un 1% de tolerancia el tiempo que tarda en ejecutarse el algoritmo puede llegar a reducirse a la mitad. También obtenemos el dato de que con las imágenes reducidas el tiempo es muy inferior. Este último resultado era predecible, ya que el tiempo que tarda el algoritmo está relacionado con los puntos que tienen las imágenes.

TestColorAccuracy (%)	2	3	4	5	6	7	8	9	10
Original - rtol = 0	58,27	72,65	81,01	85,00	87,12	88,16	89,18	89,38	89,74
30x40 Images - rtol = 0	55,29	66,86	71,34	73,65	74,81	75,89	76,25	77,10	77,22
Original - rtol = 0.01	58,21	72,26	80,27	84,54	86,41	86,50	87,06	87,83	87,86
30x40 Images - rtol = 0.01	55,17	66,45	71,23	72,84	73,79	74,34	75,55	75,67	75,68
Original - rtol = 0.1	57,69	69,82	72,12	72,10	72,83	74,46	74,85	75,36	75,53
30x40 Images - rtol = 0.1	54,78	58,97	60,84	62,77	63,45	64,70	65,66	66,50	66,88
Original - rtol = 1	50,04	55,60	57,23	59,04	60,10	61,84	62,47	64,20	64,73
30x40 Images - rtol = 1	38,73	44,11	46,17	48,60	49,54	52,22	53,13	54,77	56,25



Nuevamente los resultados son los esperables, tanto para el efecto de la tolerancia como para el comportamiento de las imágenes reducidas. Al reducir las imágenes los colores que son combinación de los principales (RGB) empiezan a ser confusos debido a que todo está más comprimido y se pierda calidad de imagen. Esto se ve reflejado en los datos, dónde todos los experimentos realizados con las imágenes reducidas siempre presentan una menor tasa de acierto para las mismas condiciones, alrededor de un 10% menos de aciertos. Al contrario que KNN, usar imágenes reducidas para encontrar los colores no parece una buena práctica de forma general.

A lo que refiere a la tolerancia podemos observar que la única que deberíamos considerar, dentro del conjunto estudiado, es la del 1%, ya que el resto de las tolerancias son demasiado grandes y perjudican la eficacia del algoritmo. Utilizando una tolerancia relativa de 1% conseguimos reducir mucho el tiempo de ejecución y la tasa de acierto prácticamente no se ve reducida.

Conclusión global

Hemos observado el comportamiento de nuestros algoritmos y podemos afirmar que, para una aplicación real, como sería un etiquetador automático de imágenes en una tienda online, tendríamos que mejorar mucho estos algoritmos ya que su tasa de éxito tendría que ser mayor, pero a pesar de no ser perfecto, nos ha ayudado de manera didáctica. Gracias a este proyecto, hemos podido ver implementados algoritmos que solo habíamos estudiado de forma teórica y profundizar nuestros conocimientos sobre estos, tanto a nivel de funcionamiento como de rendimiento.

Otro punto positivo de este proyecto es que nos ha ayudado a mejorar como programadores y a aprender más sobre Python, ya que hemos aprendido a utilizar nuevas librerías muy útiles y a programar de una forma más eficiente.

Como opinión personal, creemos que sería positivo llegar a crear una aplicación funcional y no solo quedarnos con los algoritmos, ya que de esta forma aprenderíamos más sobre programación y observaríamos como se traslada todo lo aprendido en la asignatura a un proyecto real que podríamos usar al día a día.